NPC Client Script Syntax

Document version: 0.1
Status: Draft
Last Modification: 2004.12.16

Created by:
Keith Fulton

Other authors:

Table of Contents

History

2004.12.16
: Keith Fulton
: Initial version

SuperClient Overview

PlaneShift will one day support many thousands of players and many thousands of npcs running on the same instance, in the same "world". However, to do this will take a relatively large number of game servers talking together and maintaining consistent world state among them. This has a high requirement for donations and puts a heavy burden on the developers to maintain and administer many boxes.

Until that long-term vision is achieved, though, it is desirable to run for as long as possible on one server, then scale to two nicely and so forth. But the top priority is to get as many people onto one server as we possibly can.

Another high priority is to make sure that the NPC AI system achieves the following goals:

- Allows for multiple versions of AI to be implemented, or to evolve over time.
- Has maximum independence from the core game server code. To the game server, a Player Character with a human being at a computer and a Non-Player Character which is automated should be as similar as possible.
- Takes into account the key objectives for the Game Servers, which are to maximize number of players allowed to play on a single box and to not assume unlimited server power, or even local access to multiple servers. If servers are donated to the team to help the cluster, it is not reasonable to assume they will all be at the same data center, or even in the same country.

For these reasons, the PS team has developed a 'superclient' architecture for NPC management. A 'superclient' is a process which connects to the Game Server using the same basic protocols and account structures as a player client. However, the superclient is managing multiple entities in the game instead of just the single player entity managed by a normal 3d client. The superclient has a special message protocol oriented around controlling multiple things at once which packs messages more tightly and makes networking more efficient. On the Game Server, though, there is almost no difference between a PC and an NPC outside the NPCManager which handles and unwraps these special network messages for the rest of the server.

The first superclient written is the one in CVS called 'npcclient', which the remainder of this document will focus on. Its focus is on stupid, but numerous monsters that are efficient on CPU to manage so a single instance of npcclient can possibly manage 500-1000 npcs concurrently.

Another document will be produced to explain the exact network protocol used by psserver and npcclient so that new superclients with their own AI and own scripting abilities/languages can be implemented. Such is the design of psserver that we can accommodate not only multiple superclients attached concurrently, but multiple different AI implementations with different levels of sophistication and scalability as long as both comply with the npc network protocol.

The best example of this is that npcclient is designed to have the bare minimum intelligence for attackable monsters, but to be able to support as many as 1000 of them concurrently at any one time. This takes a lot of sophiscation on the scripting side, managing 1000 multitasking event-

driven scripts for 1000 monsters, but each one cannot be very smart because our CPU budget for each monster is 1msec/sec.

An alternative superclient might be for very smart, very rare dragons and require 100% CPU on a separate box just to run the AI for 2 dragons, instead of 1000 orcs.

Network Protocol Concepts

Any AI system implementer will need to understand what information his creatures will have available to them and when, so he/she can work within that system to determine behavior. This section will not be a detailed explanation of each networking message, but instead an overview of the types of information npcclient gets from the game server and sends to the game server.

Perceptions

The most important thing to understand is the superclient event model. The Game Server notifies all superclients of events relevant to them, and it is up to each superclient to decide when/how/if to react to each event. This constant stream of events, along with periodic updates of entity positions, represents all the info each superclient knows about the world from the game server. (Superclients of course track their own information as well.)

The following is a list of perception events the server sends to superclients such as npcclient, together with their purpose and what a superclient might use them for.

Time of Day

The Game Server sends a Time perception once per game hour to all superclients, with a number from 0 to 23 indicating the hour. If a superclient needs more fine-grained time reactions, the superclient can generate time perceptions of its own in between and react accordingly.

This perception allows npcs to change behaviors or state depending on the time of day. For example, you could have nocturnal hunting creatures who sleep during the day, or npc city merchants who are in their shops during the day and at home at night.

Talking

If a player attempts to talk to an NPC, the Game Server handles the dialog aspect of this event. It was viewed as beneficial to have one central database of dialog for NPCs, and as not beneficial to have multiple implementations of dialog algorithms across superclients, so this is all centralized.

However, NPC behavior surrounding this dialog is entirely determined by the superclient. The Game Server will never move an NPC or make an NPC animate without being told to by the superclient managing that NPC.

This Talk perception message allows the superclient to turn a city merchant to face the speaker, or to make the Orc attack if spoken to, etc. To assist with different reactions depending on the player, the worst mutual faction score between player and NPC is sent with this perception.

Attack	If a player attacks an NPC, his superclient is immediately notified even though it may take 2-5 seconds for the first swing of the weapon to land. This isn't strictly fair but can help compensate for the lagtime to the superclient and will allow fast-fighting NPCs to hit first even when attacked.
Group Attack	If a player is a part of a group, NPC tactics will change slightly. The NPC may not want to strictly always fight back against the melee fighter, but may decide to pick out the wizard in the group, or the healer or the rogue. This perception sends the list of entities in the attacking group and includes a specifier on each one of what their single highest ranking skill is, to give the NPC some idea of what he is up against.
Damage	Whenever the HP of an NPC goes down, whether through melee, fall damage, spells hitting him or whatever, the superclient managing that NPC gets this perception. The entity is specified (if any) that caused the damage and the amount of HP lost are both sent as part of the perception. NPCs can use this to manage their hate lists or help them decide who to attack and how to fight back, or run to escape, etc.
Spell	If a spell is cast by any player, the perception of it is sent to all superclients. The perception should be broadcast internally in the superclient to any npc that is within 20m of the spell caster. Each perception comes with the category of the spell, as defined in the spells table in the game server database, the severity of the spell (also defined in that table) as a number between 0 and 25, and the caster and target entities. Each NPC evaluates the caster and target to decide on friend or foe, then decides how to react to that spell type, given its severity. For example, a direct damage spell cast on the NPC himself may not require any further reaction because the NPC will receive a Damage perception covering the same event. If the caster heals her groupmate though, and her groupmate is fighting the NPC, this could make the NPC hate the caster more even though the spell did not directly affect the NPC.
Death	If a player or an npc dies, the Game Server notifies all the superclients about this. The superclient managing the now-dead npc must shut down his AI and stop him from moving. If the dead entity is a player, the npc may want to remove the player from his hatelist or stop attacking and so on.

This list is expected to grow somewhat over time, but these are the basics and it probably shouldn't get too long. NPCClient also makes use of the perception system to generate its own perceptions that do not come from the Game Server. These are primarily things which are CPU-intensive to determine and beyond the budget of the Game Server NPC manager to handle.

NPCClient – Types vs. NPCs

NPCClient is a program designed to handle as many as 1000 npc's concurrently. Since NPCs are also scripted in xml rather than programmed in C++, this means that a naïve implementation would require 1000 scripts for these 1000 npcs.

In this implementation, however, npc scripts are designed to be shared across multiple NPCs. Each NPC in npcclient is assigned to an "NPCType," which is a set of behavior scripts and reaction scripts which determine how the npc will walk, talk, fight or act like an animal. This feature makes for excellent scalability and productivity of the npc scripter, but means certain aspects are more complex. The biggest example of this is the divide between npc-specific information and data that is shared at the NpcType level.

For example consider this NpcType script:

```
<npctype name="Wanderer">
    <behavior name="walk" decay="0" growth="0" initial="50">
        <move vel="1" anim="walk" />
    </behavior>
    <behavior name="turn" completion_decay="100" growth="0" initial="0">
        <rotate type="random" min="90" max="270" anim="walk" vel="30" />
    </behavior>
    <react event="collision"          behavior="turn" delta="100" />
    <react event="out of bounds"      behavior="turn" delta="100" />
</npctype>
```

There will be a lot more detail on how this script works in later sections, but for now just note that this basic type of NPC will walk and he will turn, and that events of collisions and being out of bounds will make him turn. Wandering at its most basic level, is these two behaviors.

Now consider these two npc definitions:

```
<npc id="4" type="Wanderer" name="Scary Orc" region="dark forest" />
<npc id="5" type="Wanderer" name="Bambi"     region="sunny meadow" />
```

Each of these NPCs defined here is a "Wanderer", but one wanders in the "dark forest" while the other wanders in the "sunny meadow". The region defining the "out of bounds" condition is named at the npc level even though the behaviors are shared.

In general, this type of separation of situation-specific data and general archetype behavior information goes on throughout npcclient scripting. The more you, the npc scripter, understands and leverages these differences the less scripting you will have to do and the less maintenance you will have to do to add new npcs, change maps or add new behaviors to existing ones.

NPCClient – Behavior Scripting

A “Behavior” in NPCClient is a script to run to make the npc behave a certain way. It is almost like a state, for AI designers used to state machine npcs. Example behaviors might include “guard patrol”, “wander in forest”, “fight attackers”, “smith an item”. Before we talk about how everything fits together we will go over what is possible in npcclient with these detailed behavior scripts. The following is a list and reference description of each behavior script operation or command you can use.

Move Operation (Net load: 0, CPU Load: 5)

The most basic behavior operation in npcclient is <move>. It simply tells the npc to move forward at a certain speed in the direction he is currently pointing, and to play a named animation while doing so.

Ex:
```
<move vel="1" anim="walk" />
```

This operation has no preset end or duration, so it will go until the behavior is preempted by another higher priority behavior such as the one to turn, triggered by a collision perception. (See the next section for a more detailed explanation of how behaviors pre-empt each other and change priorities.)

This operation has low networking overhead since all it does is trigger a single Dead Reckoning (DR) update to start the forward motion. It does have CPU overhead during the entire duration of it running though, since there is no way for npcclient to know what it might hit along this path. Every 500msec this operation wakes up and checks its latest movement for collisions, generating a perception called “collision” for the behavior script to react to. Also at each 500msec checkpoint, it checks the owner NPC’s region setting (such as “sunny meadow” in the example above), and if there is a region specifed, the operation checks its current position to make sure the NPC is still “in bounds”. If the NPC has made it outside the region in the last half second, it fires a perception called “out of bounds”. It will not fire another “out of bounds” perception until an “in bounds” perception has been detected.

MovePath Operation (Net load: 0, CPU Load: 0)

This operation is another very easy one: <movepath>. It tells the npc to set its preset path to a 3d spline specified in another file by name. Each client will take care of playing the right animations along the spline as the npc progresses. The intent of this operation was really simply for birds to be able to fly through the air as decoration rather than any gameplay reason. Aside from birds, most creatures do not follow 3d spline paths in their normal behaviors.

Ex:
```
<movepath path="bird1" />
```

This operation has no present end or duration, and will actually loop if not stopped by another behavior.

This operation has almost no network overhead, as the packed bytes of the spline are sent 1 time across the network to anyone needing to see this npc, and no other network activity happens until the <movepath> operation is interrupted.

This operation also has almost zero CPU overhead, because there is no collision detection and no bounds checking on this type of movement. The reasoning is that since predefined paths are being followed, it is up to the designer/scripter to ensure that the path goes where they want it to and does not hit anything along the way.

MoveTo Operation (Net load: 0, CPU Load: 0)

This is another very simple operation: <moveto>. It tells the npc to turn and face a certain point in 3d space, then move in a straight line to that location at a certain speed. The turn is not animated, so if you want an animated turn, you need a <rotate> op in front of this (see below). <moveto> also specifies an animation name to play while performing this move.

Ex:
```
<moveto x="-38" y="0" z="-169" anim="walk" />
```

When this operation is executed, the angle necessary to point in the right direction towards this 3d coordinate is calculated and the DR message to start the anim, start the movement and set the rotation angle so it will look correct is sent. Then the operation calculates the time it will take to traverse the distance at the specified speed and suspends itself for that length of time before waking up and completing. Thus, this operation has zero processing during these ½ second update opportunities. No CD checking is done and no bounds checking is done.

When the operation wakes up a few seconds later, the <moveto> is considered complete, and npcclient sends another DR message to stop the movement.

While this operation is very cheap, it is also not very flexible because the coordinates are in the NPCType and not in the NPC. Thus any NPC using this NPC Type with this command in it is going to have to be willing to move to this exact 3d coordinate.

Locate Operation (Net load: 0, CPU Load 0)

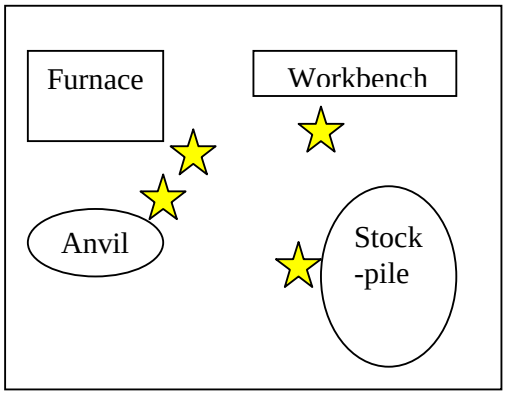
This operation is one of the more flexible and crucial operations in npc scripting. It allows the scripter to have his npc "find" a certain thing or type of thing, and remember that location for other scripting commands later such as turning, moving or attacking.

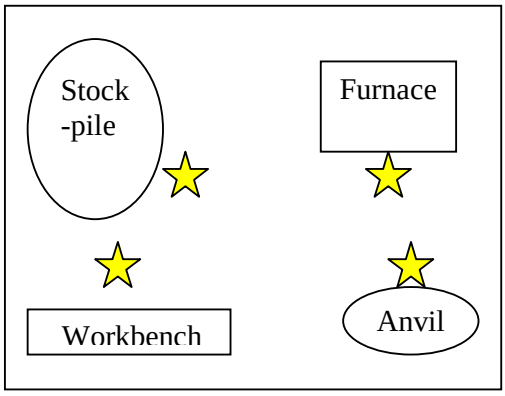
Ex:
```
<locate obj="perception" />
<locate obj="enemy" />
<locate obj="furnace" range="50" />
```

Example 1 finds the last perception received by the current npc and sets the active target location to the location of that perception. This allows npcs who receive "Talk" perceptions to turn to face the person talking to them. It allows npcs who have a spell cast on them to turn to face the caster, and so on.

Example 2 finds the location of the currently active “enemy” of the NPC, which is the entity with the highest score on the NPC’s hate list, and sets that location as the active target location for other operations. This allows NPCs to turn to face the person they are attacking, chase attackers who are retreating, etc.

Example 3 uses predefined lists of named location types to find the closest one to the npc and sets the active target location to the coordinates specified there. This is the most subtle of the 3, and in some ways the most powerful so it deserves more discussion. Consider the following pictures:





Here we have two blacksmith shops with similar items but different layouts. They might be side by side or they might be in entirely different cities. LocTypes specify a list of Anvil locations, or a list of Furnace locations, as marked by the stars on these pictures. When Example 3 says to locate the nearest furnace, an NPC in smithy #1 will find his furnace while the NPC in smithy #2 will find his own furnace. They will be able to share one script for moving between these points and acting like a real blacksmith in their own location, even though they are in totally different smithy shops.

This will become even more important when we implement player housing and player shops. It will be possible with the scripts we have already today for a player to build the blacksmith shop of his own design and for him to hire an npc smith to come work in that shop. The Smith NPC he hired will be able to “know” how to get around the player’s shop magically.

Something intended to be supported here but not actually implemented yet was the ability to locate named objects or entities as well, so an NPC could locate a sword someone dropped on the ground and things like this. Better integration with the entity system and perhaps integration with data from the maps themselves rather than externally generated lists of points could make this even more powerful.

These commands take almost no CPU power and do not use the network at all.

Rotate Operation (Net load: 1, CPU Load: 0)

This operation <rotate> can be used to turn to a certain angle, to turn a relative amount, to turn a random amount or to turn to face a previously located object. This turn is animated, to give a more realistic look.

Ex:
```
<rotate type="random" min="90" max="270" anim="walk" vel="30" />
<rotate type="locatedest" anim="walk" vel="90" />
```

The first example here is a random turn of between 90 and 270 degrees from the current heading, with an angular velocity of 30 degrees per second, while playing the "walk" animation. 30 degrees per second means that the maximum turn of 270 degrees will take the npc 9 seconds.

In the second example, we have already used our <locate> operation to find a point of interest near us, and we want to turn to point directly at it. Here a different (faster) angular velocity is specified but other angles aren't necessary because they are implied by the current position of the NPC and the located special destination.

This operation requires a DR update to start and stop the rotation, as the only networking overhead. CPU overhead is also minimal since npcclient calculates the time required to turn the full amount and puts this script to sleep for that many msec until the turn is done, then stops it.

Navigate Operation (Net load: 1, CPU Load 0)

The <navigate> operation is a placeholder right now and is simpler than it will be in the future. It is designed to work with <locate> and it moves the NPC in a straight line to the target location found by the <locate> operation.

Ex:
```
<navigate anim="walk" />
```

The only thing the operation specifies is the animation to play while moving. It works much like <moveto> but uses the locate target destination instead of a single fixed location. The only network overhead are the Start and Stop DR messages, and the only CPU overhead is calculating how long to sleep between starting and stopping.

The intent of this command in the future is to have more advanced pathfinding techniques and information in it, so the movement does not have to be in a straight line. If the smith is standing in the smithy working, and the time perception tells him to go get a drink, he should be able to <locate obj="tavern"> and then <navigate> through the entire city to find and enter the tavern. This is not implemented yet as it requires a major data effort and finalized maps.

Chase Operation (Net load: 3, CPU Load 3)

Chasing a player is one example where both turning and moving at the same time are required to give a realistic effect to the behavior. We have <move> and <rotate> but this one does them together and intelligently to follow a targeted player or entity. Normally, this will be used to get npc's to chase after players if players run away during a fight.

Ex: `<chase type="enemy" anim="walk" range=”2” />`

In this example, the npc will play the animation called “walk” while moving and turning as appropriate to reduce his range to his most hated enemy. Of course, he is limited by his speed and if the player can run faster than he can chase, he will never catch the player.

The “type” attribute of the operation can be either “enemy” as shown here, “owner” to follow the NPC's owner around, or “nearest” to chase after the nearest player at the time the operation was initiated. This operation will continue until interrupted by another behavior or until the distance to the target from the npc is less than the range specified. (2 is the default if no range is specified.)

This operation sends DR messages whenever it turns or changes course, plus the DR messages to start the forward motion and animation, and to stop them. The CPU load is somewhat heavy because CD must be performed as the npc is chasing, since his path is not predetermined.

Wander Operation (Net load: 2, CPU Load 1)

Using random <move> and <rotate> operations works well for creating randomized wandering behaviors in open, outdoor spaces and scripters should be using that approach there. However, in tighter, indoor areas like the hydlaa sewers or the dungeon, a more specific type of randomness is required.

The <wander> operation makes the npc in question walk between an independently defined set of “waypoints”. Each waypoint is a 3d coordinate and a tolerance radius. The radius adds a ‘fuzzy’ factor so that multiple npcs sharing the same set of waypoints will not all walk to the exact same centimeter location. Even in tight locations, a tolerance radius of .5-1m will be a lot better than 0m.

Ex: `<wander anim="walk" />`

As you see, the only thing specified by the wander operation is the animation to play while wandering. The NPC starts by walking to the nearest waypoint to its current location, then follows the waypoint choices to decide where to go next. An example of waypoint definitions is here:

```
<waypoints>
  <waypointlist>
 <waypoint name="p1" x="-45.56" y="0" z="-158.87" radius="1" allow_return="y" />
    <waypoint name="p2" x="-49.95" y="0" z="-152.48" radius=".5" />
    <waypoint name="p3" x="-57.32" y="0" z="-142.02" radius="1" />
    <waypoint name="p4" x="-66.51" y="0" z="-149.06" radius="1" />
    <waypoint name="p5" x="-57.81" y="0" z="-160.12" radius=".5" />
    <waypoint name="p6" x="-54.07" y="0" z="-162.97" radius="1" />
  </waypointlist>
  <waylinks>
```

```
    <point name="p1">
      <link name="p2" />
      <link name="p6" />
    </point>
    <point name="p2">
      <link name="p3" />
      <link name="p5" />
    </point>
    <point name="p3">
      <link name="p4" />
    </point>
    <point name="p4">
      <link name="p5" />
    </point>
    <point name="p5">
      <link name="p6" />
    </point>
  </waylinks>
</waypoints>
```

This XML looks complex but in fact is fairly easy to export from a tool like 3ds max. First, a list of all the waypoints is defined, with the <waypoint> tags. Each has the aforementioned 3d coordinate and the tolerance radius, plus a name. The first waypoint "p1" also has an optional attribute called "allow_return" which means that if the NPC has just come from p1 (and thus is at p2 or p6 now) and the NPC is deciding which waypoint to go to next, he is allowed to go back to p1.

Then the "waylinks" section defines which points are connected to each other. Through this network of points and links, you can lay out a structure similar to your maps and the NPCs <wander>ing will be able to navigate them without bumping into walls, etc.

This operation is very light on the network because it is all straight-line navigation between known, preset points. There is 1 DR update per point hit by the npc. It is also quite light on CPU power requirements because the points are known, which means no collision detection has to be done. One tricky thing about this is that normally non-horizontal walking is accomplished on the client with collision detection with the ground/stairs. In order to get the most accurate-looking result on the client, the client DOES do collision detection on these npcs. However, npcclient does simple linear interpolation between the starting and ending y-coordinates of the 2 waypoints as an approximation. Thus the server and client will not have the identical y-coordinate in all cases. This should almost never be noticeable, though.

Melee Operation (Net load: 1, CPU Load: 2)

The <melee> operation handles starting and ending fights for the NPC. It finds the most hated enemy in range and tells the server to attack him. If there is no enemy in range, it finds the nearest enemy that it does hate and prepares to chase him.

Ex: `<melee seek_range="10" melee_range="2" />`

The "melee_range" attribute specifies how far away the enemies can be while still fighting without the npc needing to move. 2 meters is the maximum of how far away a player can be to hit the npc also, so in most cases this is a good value. The "seek_range" specifies how far around himself the npc should look for other enemies if none are found within the melee_range. If there are no

enemies within the seek_range, the npc starts to calm down and his current behavior (making his 'melee' operation active) is lowered in priority. If an enemy is found within the seek_range, the NPC's active enemy is set to this found person so the <chase type="enemy"> operation can run after him.

This operation's only network load is informing the server of its melee target and attack mode, and switching these occasionally as people die, etc. The CPU load is in periodically (2x per second) checking the hate list for the NPC to make sure the most hated person is being fought. If no one on the hate list is within the melee range, a more expensive operation is done to find any hated people within the seek_range, but this operation isn't performed very often.

The NPC Hate List is a very important thing for an npc scripter to understand. Each NPC maintains its own list of entities that it does not like, and each of these entities gets a hate score based on what the entity does to the npc. For example, an npc on startup has an empty hate list. But then a player ("Androgos") attacks him and hits him for 10 damage:

Hatelist: Androgos-10

Now Androgos is the most hated player (because he is the only one) and the npc chooses to fight him back with his <melee> operation. A few seconds later in the fight, Androgos hits him again for 12 damage and Androgos's friend "Gronomist" casts a Fire spell on the npc also, for 25 damage.

Hatelist: Androgos-22, Gronomist-25

Next time the <melee> operation is checked, Gronomist becomes the most hated person and the npc will switch his attack to focus on Gronomist. Meanwhile, a 3rd player in the group, "Djagg" is the healer of the group and heals Androgos's wounds for 15 HP, which the NPC really hates.

Hatelist: Androgos-22, Gronomist-25, Djagg-45

So now the NPC decides to go after the healer. And so on. The idea is that the tactics of the NPC can be determined by the way he reacts to different events during combat such as spells, damage and healing. Different reactions can be scripted for different npcs, as we will see in the next section.

When a player dies, they are removed from the hatelist. So the NPC will not recognize the reincarnated player when he comes back. This keeps the entire world from attacking players all the time, and keeps hatelists reasonably short. Likewise, when the NPC dies, his entire hatelist is cleared out.

Pickup Operation (Net load: 0, CPU Load: 0)

This operation allows an NPC to pickup an item he finds on the ground, and optionally equip it if he can use it. This operation is not yet implemented.

Drop Operation (Net load: 0, CPU Load: 0)

This operation allows an NPC to drop an item he has in inventory on the ground. Not yet implemented.

Loop Operation (Net load: 0, CPU Load: 0)

This operation allows you to loop a section of your script a certain number of times. This is mostly useful for city npcs such as smiths and tavern barkeeps. Behaviors in general will loop if not replaced by another behavior, but if you want subloops within the script, this is an easy way to do it.

Ex:

```
<loop iterations="20">
  <locate obj="fire" range="10" />
  <navigate anim="walk" />
  <wait anim="stand" duration="30" />
  <locate obj="anvil" range="10" />
  <navigate anim="walk" />
  <wait anim="hammer" duration="20" />
</loop>
```

In this example, the blacksmith is scripted to walk back and forth between his fire and his anvil 20 times before moving on to the next operation in his script to act like a blacksmith.

This operation does not affect the network at all and does not have CPU overhead.

Wait Operation (Net load: 0, CPU Load: 0)

This operation makes the NPC wait a specified length of time before progressing to the next operation. It will loop an animation during this time also for you, so you can use it for crafting actions, guard monitoring, or whatever you need that is a timed animated activity.

Ex: `<wait anim="stand" duration="30" />`

This example tells the NPC to play the “stand” animation and do nothing for 30 seconds. This requires a network DR message with the specified animation, and another to stop the animation when the duration is over. There is almost no CPU required because the script goes to sleep for the duration and is woken up automatically at the end.

As with any other scripted operation, this operation can be interrupted if another behavior preempts this one from completing. So if an npc is playing the above operation, and just standing there for

30 seconds, and is attacked by someone, presumably an attack behavior would become top priority and the npc would stop waiting and attack back.

This concludes the section on the detailed operations possible with npc scripting today. Now we will discuss how these operations work together to define behaviors and how behaviors interact with each other.

Behaviors and Reactions

As discussed earlier, an NPC Type is really simply a collection of behaviors and reactions to events. At the most meta level, all of us fit this definition. ☺ How we act and how we respond to stimuli determines everything about our activities over time. Thus it seems sufficient and general to model npc behavior this way also.

Each behavior gets a script made of operations as documented in the previous sections. For now we will neglect those, however and just talk about the behaviors themselves.

Here is an example of an NPC Type's behavior/reaction list:

```
<npctype name="Wanderer">
    <behavior name="walk" decay="0" growth="0" initial="50">
    <behavior name="turn" completion_decay="100">
    <behavior name="fight">
    <behavior name="chase" completion_decay="100">

    <react event="collision"       behavior="turn" delta="100" />
    <react event="out of bounds"   behavior="turn" delta="100" />
    <react event="attack"          behavior="fight" delta="150" />
    <react event="damage"          behavior="fight" delta="20" weight="1" />
    <react event="target out of range" behavior="chase" delta="0" />
    <react event="death"           behavior="fight" delta="-1" />
</npctype>
```

This collection of behaviors and reactions is sufficient to have a basic outdoor wandering monster who will avoid collisions, stay in his boundary region, fight back if attacked and stop attacking if he dies.

Similar to the way the hate list maintains a prioritized list of enemies for each NPC, each NPC also maintains a set of priorities for the list of behaviors in its npctype. The priority list starts out with 0 as the default score, unless an "initial" attribute is specified. (One behavior should have this attribute so the npc has a place to start.) Thus, any "Wanderer" NPC will start with the following priority list:

P-List: walk-50, turn-0, fight-0, chase-0

Thus on startup the NPC will begin by walking and using the operations inside that behavior tag to do something until an event occurs which preempts the behavior by making something else higher priority. For example, if he goes out of bounds, he will get an "out of bounds" perception, which he has a <react> tag for. It says to add 100 points to the "turn" behavior priority, which results in:

P-List: walk-50, turn-100, fight-0, chase-0

So his "walk" behavior is interrupted and the "turn" behavior and script is activated. This script starts running and the npc starts turning. Turning to point back into the boundary region should only take a couple of seconds. When the script complets for this behavior, the "completion_decay" of the behavior kicks in and affects the P-List again:

P-List: walk-50, turn-0, fight-0, chase-0

There are many real-world examples of how “completion_decay” exists in humans. Many actions are the top priority until they are completed, and then one simply doesn’t need to do them anymore. “completion_decay” is modeling that change in priority when an action is performed.

Now that the priority for turn has gone back to 0, “walk” again is the highest priority behavior and the npc starts walking again.

So now the npc is attacked by a player in the forest. “attack” is a perception sent by the server, as discussed in a previous section, and one for which this npctype has a <react> tag, telling him to add 150 to his “fight” priority, resulting in:

P-List: walk-50, turn-0, fight-150, chase-0

So now the npc stops walking and starts his fight behavior, which probably has the <melee> operation in it, using the hate list to fight back against one or more attackers. As the attack progresses, damage perceptions come in. A damage perception of 10HP has a <react> script also, which adds 10 points to the fight priority and also has a “weight” attribute which affects the hate list instead of the behavior list:

P-List: walk-50, turn-0, fight-160, chase-0

Since the fight behavior is already the highest, it might be redundant to keep increasing it like this, but personally I view it as the npc getting more enraged, which means it will take him longer to calm down when he runs out of enemies. The weight=”1” attribute means that the attacker’s hate score will go up by 1*HP.

Now imagine that the attacking player is losing the fight, and decides to run away. Very quickly, the “target out of range” perception will fire, created by the melee operation when the most hated enemy is farther away than the melee_range but within the seek_range. This NPC is scripted to chase after hated enemies if they run away with his <react event=”target out of range”> tag. The “delta” specified of 0 is a special delta which means “Make the named behavior the highest no matter what.” When reactions are coded with 0 deltas, NPCs are really acting like state machines again.

P-List: walk-50, turn-0, fight-160, chase-185 (Highest means at least 25 higher than the next highest to cause a preemption and switch.)

So now the NPC stops attacking and begins chasing the most hated enemy. The <chase> tag has a range specified for the chase to be completed (probably 2m again), so when the npc has caught the player, the chase behavior completion decay hits. The chase behavior has a specified completion decay of 100, but since chase was escalated artificially high by the 0 react, it is restored intead back to its original value.

P-List: walk-50, turn-0, fight-160, chase-0

So the npc stops chasing and resumes fighting the player immediately when he catches him. Now the npc finally kills the player and the fight is over. The “fight” behavior is still the highest one, so the npc keeps looking around for other enemies to fight, such as group members. If none are found, his fight behavior keeps decreasing by 10 points per iteration.

P-List: walk-50, turn-0, fight-160, chase-0
P-List: walk-50, turn-0, fight-150, chase-0
P-List: walk-50, turn-0, fight-140, chase-0
P-List: walk-50, turn-0, fight-130, chase-0
And so on…

Eventually, “walk” behavior becomes the most active again and the npc resumes his wandering of the forest.

The key concepts here are that the NPC constantly maintains a priority list of behaviors and a priority of list of enemies to attack. <react> tags can affect behavior priority, hatelist priority or both.

The interactions of events, behaviors and reactions can become very complex. The npc can be interrupted by an attack while moving, turning or even chasing another player. This priority structure was designed to allow for rich behavior interruption and resumption while having realistic reactions to world events and activities. Hopefully, with the right scripts we can create very life-like npcs that are also efficient on networking and CPU, with emergent behaviors that stay challenging and interesting to players for a long time to come.